**Extending BCDM to cope with proposals and evaluations of updates**

Luca Anselma; Alessio Bottrighi; Stefania Montani; Paolo Terenziani. Extending BCDM to cope with proposals and evaluations of updates. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING. 25 (3) pp: 556-570.
DOI: 10.1109/TKDE.2011.170

# Extending BCDM to cope with proposals and evaluations of updates

Luca Anselma, Alessio Bottrighi, Stefania Montani, Paolo Terenziani

**Abstract**— The cooperative construction of data/knowledge bases has recently had a significant impulse (see, e.g., Wikipedia [1]). In cases in which data/knowledge quality and reliability are crucial, proposals of update/insertion/deletion need to be evaluated by experts. To the best of our knowledge, no theoretical framework has been devised to model the semantics of update proposal/evaluation in the relational context. Since time is an intrinsic part of most domains (as well as of the proposal/evaluation process itself), semantic approaches to temporal relational databases (specifically, Bitemporal Conceptual Data Model (henceforth, BCDM) [2]) are the starting point of our approach. In this paper, we propose BCDM$^{PV}$, a semantic temporal relational model that extends BCDM to deal with multiple update/insertion/deletion proposals and with acceptances/rejections of proposals themselves. We propose a theoretical framework, defining the new data structures, manipulation operations and temporal relational algebra and proving some basic properties, namely that BCDM$^{PV}$ is a consistent extension of BCDM and that it is reducible to BCDM. These properties ensure consistency with most relational temporal database frameworks, facilitating implementations.

**Index Terms**— H.2.4.m Temporal databases, H.2.3.d Database semantics, H.2.0.b Database design, modeling and management.

## 1 Introduction

Relational DBs are one of the main paradigms in data management, with a wide applicative impact. Recently, relational DBs have been adopted in order to cope with an emerging phenomenon: the cooperative construction of databases (see, e.g., Wikipedia [1]). In particular, in cases in which data/knowledge quality and reliability are crucial, proposals of update/insertion/deletion of data need to be evaluated by experts. Indeed such a phenomenon (called "proposal vetting" in the paper) often involves relational data (consider, e.g., the Citizendium encyclopedia [3], and the cooperative management of clinical guidelines [4]). Despite the generality and the spread of the phenomenon itself, until now proposal vetting has been mostly coped with in an ad-hoc way in the relational context (primarily at the application level). On the other hand, a domain- and application-independent and theoretically grounded solution should be provided once and for all, so that application developers can safely adopt it, and just focus on the application-dependent aspects of their problems[1]. Additionally, current relational ad-hoc solutions have specific limitations, such as the fact that they do not support (in data model, and in manipulation and query operations) a sound treatment of temporal phenomena. The goal of our work is to provide a general approach extending the relational DB theory to cope with proposal vetting, and overcoming the above limitation of current solutions.

The starting point of our approach is the consideration that, indeed, proposal vetting intrinsically involves the notion of transaction time [5, pages 3162-3163] (the time of proposal, of evaluation, of insertion/logical-deletion). Additionally, proposal vetting might also be applied to temporal data (as in [4] and in our running example). In such cases, also the valid time [5, page 3253] of data should be considered. 25 years of research in (relational) Temporal Databases (TDBs henceforth; see, e.g., the cumulative bibliography in [6]) have widely demonstrated that, in order to correctly manage transaction-time data (or transaction+valid-time data) in the relational context, specialized techniques must be used[2]. Therefore, the adoption of TDB techniques is necessary to cope (in data model, manipulation operation, and query operations) with proposal vetting on relational data (since at least transaction time must be coped with). A possible solution might be to extend one of the current temporal relational

• *L. Anselma is with the Dipartimento di Informatica, Università di Torino, Torino, Italy. E-mail: anselma@di.unito.it.*
• *A. Bottrighi, S. Montani and P. Terenziani are with the Dipartimento di Informatica, Università del Piemonte Orientale "Amedeo Avogadro", Alessandria, Italy. E-mail: {alessio.bottrighi, stefania.montani, paolo.terenziani}@mfn.unipmn.it*

[1] Indeed, this is a widely used strategy in Computer Science. For instance, temporal databases mainly are born to solve once and for all (and in a theoretically sound way) many domain-independent issues related to the treatment of time.

[2] "Two decades of research into temporal databases have unequivocally shown that a time-varying table, containing certain kinds of DATE columns, is a completely different animal than its cousin, the table without such columns. Effectively designing, querying, and modifying time-varying tables requires a different set of approaches and techniques than the traditional ones taught in database courses and training seminars. Developers are naturally unaware of these research results (and researchers are often clueless as to the realities of real-world application development). As such, developers often reinvent concepts and techniques with little knowledge of the elegant conceptual framework that has evolved and recently consolidated…" in [7], Section "Preface", Subsection: "A paradigm shift", page XVIII. It is worth stressing that, in the above quotation, no mention is made on whether DATE columns regard transaction and/or valid time. For instance, all the problems exemplified in Section 1 of TSQL2 book about valid time [8] arise exactly in the same way if transaction time is considered.

approaches (e.g., the "consensus" TSQL2 approach [8]) to cope with proposal vetting. However, to further enhance the generality of our approach, we have chosen to extend BCDM [2], a unifying formal semantic model which has been proven to constitute the common "core" semantics of many temporal relational approaches in the literature (e.g., the approaches in [9-13], including TSQL2 [8]). In our approach (as in BCDM) the semantics of data, manipulation and query operation is modeled in a formal way, to provide a rigorous and non-ambiguous specification to implementers, and to provide a solid theoretical environment in which fundamental properties such as *reducibility* and *consistent extension* [8] can be formally proved. Thus, our work belongs to the temporal-database research and, specifically, to the stream of approaches extending BCDM to cope with new phenomena, such as periodic data [14], telic events [15], and different forms of temporal indeterminacy [16].

Coping with proposal vetting requires radical extensions to the BCDM model. Some operations (e.g., updates) must be "delayed", waiting for an evaluation which rejects them or makes them effective on the reference data model. Thus, two types of data need to be supported: the reference (accepted) data and the proposed (to-be-evaluated) data. More importantly, different update proposals concerning the same piece of data must be interpreted as alternatives: at most one of the alternative proposals can be accepted and becomes effective.

Thus, we propose a new semantic model, which we call BCDM$^{PV}$ (where "PV" stands for "proposal vetting") in which alternative proposals are first-class entities, to be explicitly and directly modeled into the data model[3]. BCDM$^{PV}$ is an extension of BCDM consisting of:

(1) a new data model to cope with both accepted and proposed data, in which alternative proposals are first-class entities (Section 3);

(2) new manipulation operations to propose insertions, deletions and updates (for proposers) and to accept or reject them (for evaluators) (Section 4);

(3) new algebraic operations on the extended data model (Section 5).

In our approach, we follow the methodology proposed in TSQL2 book [8]. We specify aspects (1), (2) and (3) in a formal way, so that the necessary properties of reducibility and consistent extension can be proved ([8, 17]). Specifically, we have proven that:

(i) BCDM$^{PV}$ data model is reducible to BCDM one;

(ii) BCDM$^{PV}$ manipulation operations are a "proposal vetting" consistent extension of BCDM ones;

(iii) BCDM$^{PV}$ algebraic operations are reducible to BCDM ones.

The fact that the above properties hold guarantees three main advantages for our approach:

(a) generality, since BCDM$^{PV}$ extends the core semantics of several TDB approaches;

(b) implementability, since, given (a), our approach can be implemented as an additional layer on top of any TDB approach based on the BCDM semantics (including TSQL2; it is worth noticing here that Oracle™ Database, since version 10g, supports both transaction time and valid time consistently with BCDM [18]);

(c) interoperability with TDB approaches based on BCDM.

*Reducibility* guarantees that the semantics of simpler operators are preserved in their more complex counterparts [8, page 233], and, together with the *consistent extension* property (elsewhere called "*temporal upward compatibility*" [17]) is needed to grant the compatibility with pre-existent approaches [17], and, thus, interoperability. Interoperability is a "sine qua non" feature for all temporal extensions to relational DBMS, to guarantee the possibility of maintaining and operating on pre-existent data. Observe that, in turn, BCDM-based approaches are reducible to the standard relational model, so that interoperability with the relational model also will hold for the implementations of our approach.

Finally, it is worth stressing that, as in BCDM, we operate at the semantic level only, so that, consistently with the explicit aims of BCDM itself[4], in this paper we do not address issues such as complexity, query optimization, integrity constraint support, storage optimization and data indexing; schema versioning is not considered, too. However, in Section 6, we briefly mention a prototypical implementation of our semantics on top of TIMEDB [19], a TSQL2-like system based on BCDM semantics that we have devised as a proof of concepts.

The paper is organized as follows. Preliminaries are briefly introduced in Section 2. Our new data model is presented in Section 3. Section 4 introduces the new manipulation operations to propose changes and to evaluate them. Section 5 introduces a new temporal algebra to query the new data model. Section 6 is devoted to describe our prototypical implementation and possible extensions. Finally, Section 7 presents related works and Section 8 addresses conclusions.

## 2 PRELIMINARIES

In this section, we introduce some basic notions for our work. At a high level of abstraction (henceforth: "process level"), proposal vetting can be described as a set of interactions between evaluators, proposers, and the database. In Section 2.1 we provide a formal description of the "process level", using Petri Nets. However, the focus of our approach is on the "data level" (e.g., what are the

[3] Of course, one could write some application software to simulate alternatives on top of BCDM or of the standard relational model, interpreting some sets of tuples as alternative. But this would be an ad-hoc ***implementation,*** i.e., application software which super-imposes a different interpretation onto a basic data model in which all data are interpreted in a *conjunctive* way. This is not what we want in our semantic approach: the basic semantic model must be extended in order to be directly able to support alternative data by itself.

[4] "It is our contention that focusing on data presentation (how temporal data is displayed to the user), on data storage with its requisite demands of regular structure, and on efficient query evaluation, has complicated the central task of capturing the time-varying semantics of data. […] We therefore advocate a separation of concerns. Time-varying semantics is obscured in the representational schemes by other considerations of presentation and implementation. We feel that the conceptual data model to be discussed shortly [i.e., BCDM] is the most appropriate basis for expressing this semantics." [8, pages 185-186]

possible evolutions of data, what are the possible manipulation operations and what are their admissibility conditions). We aim at devising an extension of a temporal relational database theory to cope with proposal vetting, considering the definition of a new data model, and of manipulation and algebraic operations. The starting point of our approach is the BCDM semantic model, which is sketched in Section 2.2. Finally, Section 2.3 introduces the running example we adopt in this paper.

### 2.1 The proposal vetting process

In the following, we clarify our notion of proposal vetting by describing the workflow of processes it involves. Many formalisms have been devised to model process interactions, including workflow formalisms (see, e.g., [20]) and different variants of Petri Nets [21]. Petri Nets are bipartite directed graphs with two types of nodes: places and transitions. Places, graphically represented by circles, correspond to the state variables of the system; transitions, graphically represented by boxes, correspond to the events that can induce a state change. The arcs connect the two kinds of nodes and express the relation between states and events occurrence. In particular in our modelization we use the Well-Formed Net formalism [22] that extends Petri Net formalism with "colour", allowing one to describe the system in a more compact way.

The Well-Formed Net model in Fig. 1 describes the dataflow and the user behaviour in proposal vetting. The transition T1 models the issue of a proposal. The inputs of T1 are a token labeled <k> from PROPOSERS, a token labeled <x> from DB_EVALUATORS and a token labeled <p> from DB_PROPOSERS, to model the proposal risen by a proposer, who can take into account the current status of the reference DB (i.e., DB_EVALUATORS) and of the DB of proposals (i.e., DB_PROPOSERS) respectively. The output of T1 is a token labeled <m> in the place PROPOSALS (a new proposal is risen). Additionally, tokens with the original values (represented by the fact that the labels on the input and output arcs are the same) are returned to the places PROPOSERS, DB_EVALUATORS and DB_PROPOSERS. This means

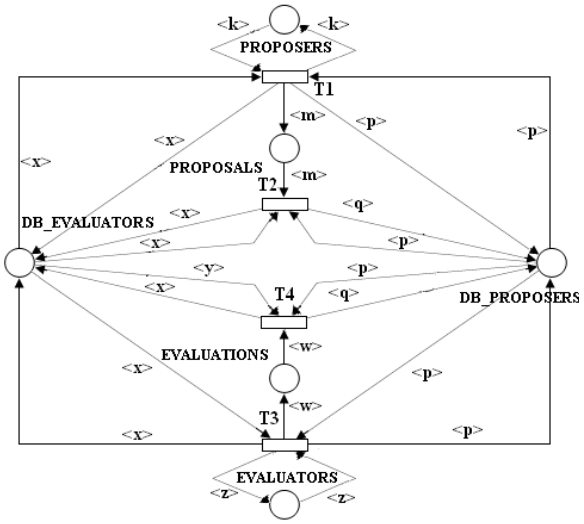


Fig. 1: the Well-Formed Net model which describes the proposal vetting phenomena.

that the content of the two databases and the set of proposers are not changed by T1. The transition T2 similarly models the storing of a proposal. Analogously, T3 models the issue of an evaluation and T4 models the storing of the effect of an evaluation.

While Fig.1 provides an abstract view of proposal vetting at the "process level", many issues, such as what are the possible manipulation operations, under which conditions they can be applied, and how data can evolve as a result of such operations are left implicit in the above formalization. Such a "data level" analysis is the goal of our approach, which is grounded on the current temporal database theory, and, specifically, on BCDM.

### 2.2 Bitemporal Conceptual Data Model

BCDM [2] is a unifying data model, isolating the "core" semantics underlying many temporal relational approaches, including TSQL2 [8]. In BCDM, tuples are associated with *valid time* and *transaction time*. For both domains, a limited precision is assumed (the *chronon* is the basic time unit). Both time domains are totally ordered and isomorphic to the subsets of the domain of natural numbers. The domain of valid times $D_{VT}$ is given as a set $D_{VT}=\{t_1,t_2,\ldots,t_k\}$ of chronons, and the domain of transaction times $D_{TT}$ is given as $D_{TT}=\{t'_1,t'_2,\ldots,t'_j\}\cup\{UC\}$ (where UC – Until Changed– is a distinguished value). In general, the schema of a bitemporal conceptual relation $R=(A_1,\ldots,A_n \mid T)$ consists of an arbitrary number of non-timestamp (*explicit* henceforth) attributes $A_1, \ldots, A_n$, encoding some fact, and of a timestamp attribute T, with domain $D_{TT}\times D_{VT}$. Thus, a tuple $x=(a_1,\ldots,a_n \mid t_b)$ in a bitemporal relation r(*R*) on the schema R consists of a number of attribute values associated with a set of bitemporal chronons $t_{b_i}=(c_{t_i}, c_{v_i})$, with $c_{t_i} \in D_{TT}$ and $c_{v_i} \in D_{VT}$. Valid-time, transaction-time and atemporal tuples are special cases, in which either the transaction time, or the valid time, or both of them are absent.

**Notation.** Given a tuple x defined on the schema $R=(A_1,\ldots,A_n, B_1,\ldots,B_l \mid T)$, we denote with A the set of attributes $A_1,\ldots,A_n$. Then x[A] denotes the values in x of the attributes in A, x[T] denotes the set of bitemporal chronons constituting the timestamp of x, $x[T_v]$ and $x[T_t]$ denote the valid and transaction time of a valid-time and transaction-time tuple respectively. ♦

**Notation.** A bitemporal BCDM tuple x is *current* if it is present at the current time ("now") in the database (i.e., it has not been updated or deleted yet). ♦

BCDM is a *semantic* and *formal* approach, for which the essential properties of *uniqueness of representation* and *reducibility* to the standard non-temporal algebra have been formally proved. This grants for the advantages discussed in Section 1, and, in particular, for interoperability with standard non-temporal relational databases.

### 2.3 A running example

In the following, we introduce a running example regarding the history of some European countries. We consider a session of work in which proposals are issued, and evaluators accept or reject them. Such a session could be a session with Citizendium [3], a collaborative encyclope-

dia where an entry can be proposed and modified by multiple authors. Citizendium improves Wikipedia in the sense that it stresses reliability, so that each entry must be approved by an editor.

We use two relations: INDEP, describing when a nation is independent and its government type, and CAPITAL, specifying the capital of a nation. All the relations have a transaction time, to model the times when proposals are issued, accepted and/or rejected. Moreover both relations have a valid time: in INDEP the valid time represents the time when a nation is independent and in CAPITAL when a city is the capital of a nation.

In the following, we introduce a session of cooperative work aimed at exemplifying the proposal vetting process. The working session is introduced as a sequence of steps:

*Step 1.* Proposer $p_1$ proposes to update the relation CAPITAL, to state that Cracow was the capital of Poland between 1040 and 1595;

*Step 2.* Proposer $p_2$ proposes the same proposal issued by proposer $p_1$ at step 1;

*Step 3.* Proposer $p_1$ proposes to modify INDEP to state that Poland has been a dictatorship until 1595;

*Step 4.* Proposer $p_3$ proposes to further update the proposal at step 3 to modify the form of government of Poland from dictatorship to republic;

*Step 5.* Evaluator $e_1$ rejects the proposal at step 4;

*Step 6.* Proposer $p_2$ proposes to update the proposal issued by proposer $p_1$ at step 3, by adding to such an update also the update (from dictatorship to monarchy) of the form of government;

*Step 7.* Proposer $p_1$ proposes to update the original version of the tuple about Poland in INDEP changing its form of government from dictatorship to monarchy;

*Step 8.* Evaluator $e_1$ queries the database to check all the current proposals concerning the independence of Poland when its capital was Cracow;

*Step 9.* Evaluator $e_1$ accepts the proposals issued by proposer $p_1$ at step 1 and by $p_2$ at step 6.

In Fig. 2 we show the evolution of our data model at different transaction times (to be explained in the rest of the paper). In the figure, at transaction time *i+1* we show the effect of the execution of step *i*. The transaction time starts when the tuples are entered into the database. The value "UC" denotes the fact that the tuple is still present (not logically deleted) in the database. For the sake of brevity, the temporal attributes are shown in a compact way. E.g., in Fig. 2(A), $\{e_1\}\times[1,UC]\times[1025,1039]$ stands for the set of triplets $\{(e_1,1,1025), (e_1,1,1026), \ldots, (e_1,1,1039), \ldots, (e_1,UC,1025), (e_1,UC,1026), \ldots, (e_1,UC,1039)\}$, meaning that the tuple has been inserted by evaluator e1 at transaction time 1 and has not been deleted yet, and that its valid time starts at 1025 and ends at 1039.

It is worth noticing that, in the example, bitemporal data are involved. Indeed, while the treatment of transaction time is an intrinsic part of coping with proposal vetting, valid time is "orthogonal" to the proposal vetting process. In fact, valid time – representing "when" the data in the relational tuples hold – is completely independent of proposal vetting. As a consequence, while transaction time is mandatory in our approach, valid time is optional. But, obviously, in case proposal vetting has to be applied on valid-time data, both temporal dimensions have to be taken into account. In the rest of the paper, for the sake of generality we focus on the treatment of bitemporal (transaction-time + valid-time) data. However, it is worth stressing that (as in the case of the BCDM model), only minor simplifications to the approach are possible in case only transaction time has to be coped with.

## 3. Extending the Data Model

To cope with the issues outlined in Sections 1 and 2, in our data model we need to distinguish between accepted data and proposals that still need to be evaluated. To this end, we introduce a two-layered approach, in which: (1) we define two categories of users: a set of proposers, who issue proposals, and a set of evaluators, who can accept/reject them, and (2) we split the data in two levels: evaluator data level and proposer data level. Namely, all validated data, accepted by evaluators, are stored in the evaluator data level. Current data in the evaluator data level constitute the reference (accepted) version of data. On the other hand, all the proposals, generated by any proposer, are stored at the proposer data level.

**Definition 3.0.1: Proposers and Evaluators**. We term Proposers=$\{p_1,\ldots,p_y\}$ and Evaluators=$\{e_1,\ldots,e_z\}$ the sets of proposers and evaluators respectively. ♦

Notice that our approach is independent of whether Proposers and Evaluators are disjoint sets or not (so that different policies can be implemented).

**Definition 3.0.2:** We define a database as a pair <**DB_Evaluators, DB_Proposers**>. DB_Evaluators is a set of relations $\{r_1(R_1),\ldots,r_k(R_k)\}$ where $r_i$ $(1\le i\le k)$ is an instance of the schema $R_i$. DB_Proposers contains, for each relation $r_i\in$DB_Evaluators, three separate sets:

**pi($r_i$)**, containing proposals of *insertion* into $r_i$,

**pd($r_i$)**, containing proposals of *deletion* of tuples in $r_i$,

**pu($r_i$)**, containing proposals of *update* (concerning tuples in $r_i$, $pi(r_i)$ and $pu(r_i)$).♦

Both in DB_Evaluators and in DB_Proposers we deal with different types of *implicit* attributes. First at all, we consider the *valid time* of tuples and with their *transaction time*. Moreover, every tuple in DB_Evaluators is associated with one (or more) elements in the Evaluators set, corresponding to the evaluators who accepted the tuple after a proposal-vetting session. Similarly, all proposals of insertion/deletion/modification are associated with one or more elements in the Proposers set. We denote as $T_e$ the attribute with domain Evaluators$\times D_{TT}\times D_{VT}$ and $T_p$ the attribute with domain Proposers$\times D_{TT}\times D_{VT}$, which extend the bitemporal BCDM attribute in order to deal also with evaluators and proposers information in DB_Evaluators and DB_Proposers respectively. We also denote with T the bitemporal attribute with domain $D_{TT}\times D_{VT}$.

**Terminology (value equivalence).** We use the term *value equivalent* as in BCDM, to denote tuples that have equal values for the explicit attributes [8]. ♦

Temporal relational semantic data models, including BCDM, usually do not admit value equivalent tuples in the same relation to support the uniqueness property for

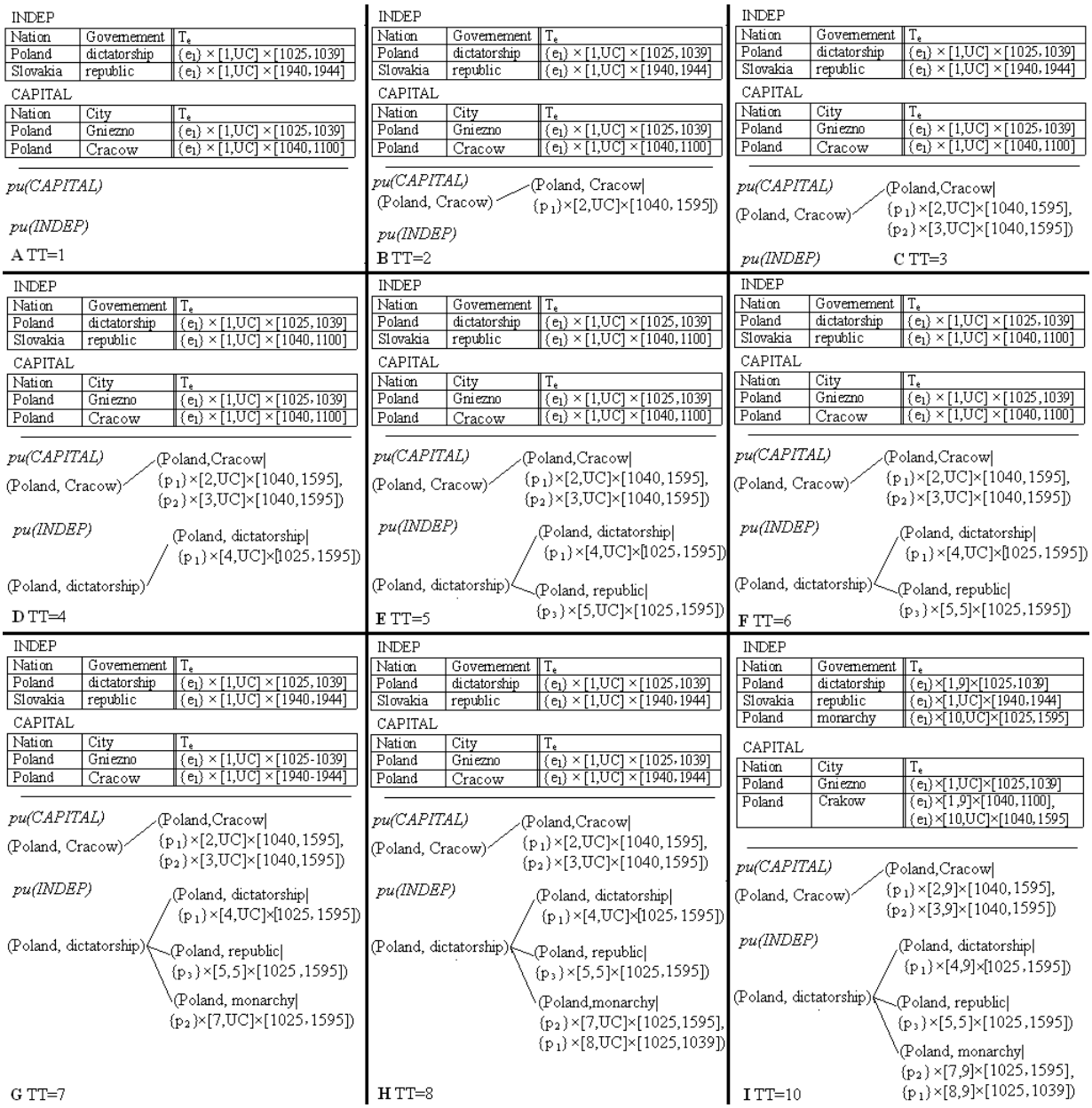


Fig. 2: A representation of our running example before the session of work (A), after Step 1 (B), after Step 2 (C), after Step 3 (D), after Step 4 (E), after Step 5 (F), after Step 6 (G), after Step 7 (H) and after Step 9 (I). In each subfigure, the upper part represents the relations INDEP and CAPITAL in DB_Evaluators (see Section 3.1) and the lower part represents the content of DB_Proposers (see Section 3.2). pu(r) contains an Update-proposal, represented – for the sake of readability – as a two-level tree. The origin of the Update-proposal is the root (on the left) (e.g., (Poland, dictatorship) is the origin of Update-proposal in pu(INDEP)) and the alternatives are its children (on the right).

their models, since "it is a major source of semantic clarity that two instances have the same information content exactly when they are identical" [8, page 221]. Since uniqueness is one of our goals (see Section 3.3), we do not admit value equivalent tuples in our data model, too.

### 3.1 DB_Evaluators

**Definition 3.1.1: DB_Evaluators.** We denote with $R=(A_1,\ldots,A_n \mid T_e)$ the schema of a relation $r \in$ DB_Evaluators, with $T_e$ defined as above. (**Condition 3.1.2**): We do not admit value-equivalent tuples in the same relation $r \in$ DB_Evaluators. ♦

### 3.2 DB_Proposers

In this section, first we briefly introduce the definitions concerning proposals of insertion and of deletion. Then we move to one of the main contributions of our approach, namely the definition of proposals of update.

#### *3.2.1 Proposals of insertion*

**Definition 3.2.1.1: pi(r).** Given a relation $r \in$ DB_Evaluators with schema $R=(A_1,\ldots,A_n \mid T_e)$, we define pi(r) as the set containing the tuples x which are proposed for insertion into r. The *schema* of pi(r) is $R'=(A_1,\ldots,A_n \mid T_p)$. (**Condition 3.2.1.2**): In pi(r) we do not admit value-equivalent tuples. ♦

#### *3.2.2 Proposals of deletion*

**Definition 3.2.2.1: pd(r).** Given a relation $r \in$ DB_Evaluators with schema $R=(A_1,\ldots,A_n \mid T_e)$, we de-

fine pd(r) as the set containing the tuples x which are proposed for deletion from r. The *schema* of pd(r) is $R'=(A_1,\ldots,A_n \mid T_{pt})$, where $T_{pt}$ represents an attribute with domain Proposers×$D_{TT}$. (**Condition 3.2.2.2**): In pd(r) we do not admit value-equivalent tuples. ♦

A tuple in pd(r) identifies the tuple in r to be deleted. Therefore, the valid time is not needed (since the explicit attributes univocally identify the evaluator tuples).

### *3.2.3 Proposals of update*

Given a relation r∈DB_Evaluators, the set pu(r) of proposals of update may concern tuples in r, or in pi(r), or in pu(r). Proposals of update must explicitly state the tuple which should be modified, and the specific changes that should be made to it (i.e., it consists of a pair <old tuple, new tuple>). In principle, each proposal of update could be modeled independently of the others. However, the underlying semantics is that all the proposals of modification concerning the same tuple must be interpreted as alternatives, since the acceptance of one proposal implicitly involves the rejection of all the others. In other words, *unlike* the standard relational model and the BCDM model, the proposal vetting context involves coping with *disjunctions* of pieces of information. We introduce a *primitive semantic notion* – the *Update-proposal* – to explicitly cope with such a new phenomenon. An Update-proposal groups together all the alternative proposals concerning a given tuple (thus resembling, e.g., the notion of *Design Object* in [23]). Defining such a grouping of alternative pieces of information as a ***primitive notion*** also provides several advantages, simplifying the definition of manipulation and algebraic operators.

**Definition 3.2.3.1: Update-proposal**. An Update-proposal may concern either (i) a tuple in an evaluator level relation, or (ii) a tuple in a proposal of insertion[5].

Given a relation schema R such that (i) $R = (A_1,\ldots,A_n \mid T_e)$ or (ii) $R = (A_1,\ldots,A_n \mid T_p)$, let r be an instance of R and x∈r a tuple in r. An Update-proposal up∈pu(r) concerning x can be defined as up = <o, Alt($alt_1,\ldots,alt_m$)>, where o=x[$A_1, \ldots,A_n$] and $alt_i$ ($1\le i\le m$) are tuples defined on the schema $(A_1,\ldots,A_n \mid T_p)$. *o* is used in order to univocally identify the tuple *x* to be updated and Alt($alt_1,\ldots,alt_m$) is a non-empty set of alternative tuples referring to the tuple *x*, representing the different alternative proposals of update concerning *x*. (**Condition 3.2.3.2**): In an Update-proposal we do not admit value-equivalent alternatives. ♦

**Terminology (type of an Update-proposal).** Given the Definition 3.2.3.1, we call the pair $<(A_1, \ldots,A_n), (A_1, \ldots,A_n \mid T_p)>$ the *type* of the Update-Proposal *up*. ♦

**Terminology (origin, alternatives of an Update-proposal).** Given the Definition 3.2.3.1, we call x the *origin* of the Update-proposal and {$alt_1$,..,$alt_m$} its *alternatives*. Since *o* is used in order to uniquely identify *x*, in the following, we call both *x* and *o* "*origin*". ♦

[5] Notice that, in our approach, proposals of update concerning a preceding Update-proposal $up\in pu(r_i)$ are directly referred to the origin of *up* (which may be either a tuple in r or in pi(r)); see the discussions in Section 4.

**Definition 3.2.3.3: origin(up) and alternatives(up).** Given an Update-proposal *up* = <o,Alt($alt_1$,$alt_2$,..,$alt_m$)>, origin(*up*) = o, and alternatives(*up*) = {$alt_1$,$alt_2$,..,$alt_m$}.♦

**Example.** Considering our running example (see Section 2.3), Fig. 2(E) shows the Update-proposals representing all the update proposals issued until step 4. The origin "(Poland, dictatorship)" in pu(INDEP) identifies the evaluator tuple to be updated. The first alternative represents the proposals issued by $p_1$ at step 3, where 4 is the transaction-time start and UC is the transaction-time end (i.e., the proposal is current), 1025 is the valid-time start, and 1595 is the valid-time end. ♦

We can finally define the set pu(r) of update proposals.

**Definition 3.2.3.4: Set of Update-proposals pu(r).** Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n \mid T_e)$, we define pu(r) (henceforth called *set of Update-proposals*) as the set containing the Update-proposals up=<o, Alt($alt_1,\ldots,alt_m$)> whose origin o identifies a tuple in r or in pi(r). The *type* of pu(r) is $<(A_1, \ldots,A_n), (A_1, \ldots,A_n \mid T_p)>$. (**Condition 3.2.3.5**): Different Update-proposals having the same origin are not admitted in the same set of Update-proposals. ♦

## 3.3 PROPERTIES OF THE DATA MODEL

In this section, we analyze the properties of the new data model. Two properties are essential: **uniqueness** of the model and **reducibility** to BCDM.

In temporal relational databases, the notion of *snapshot equivalence* has been used in order to formally characterize bitemporal relations having the same information content. To this purpose, transaction- and valid-time timeslice operators are introduced. For instance, in BCDM, $\rho^{B}_{T1}(r)$ selects from a bitemporal relation r all the tuples holding at the transaction time T1, and removes the transaction time dimension [8]. We extend such notions, to apply it also to sets of Update-proposals, and to deal with slice on evaluators/proposers.

**Definition 3.3.1: Slice operators**. Given an Update-proposal up = <o, Alt($alt_1,\ldots, alt_m$)> and a proposer p, we define the proposer-slice operator on an Update-proposal as follows:

$\pi^{PV\text{-}atv}_{p}(up)=<o, Alt(\pi^{atv}_{p}(alt_1), ..., \pi^{atv}_{p}(alt_m))>$

where

$\pi^{atv}_{p}(x) =\{z : z[A]=x[A]\wedge z[T]=\{(t,v):(p,t,v)\in x[T_p]\} \wedge z[T]\neq\varnothing\}$.

"PV" stands for "Proposal Vetting" and "atv" stands for the implicit attributes in the schema of x: proposer, transaction time and valid time. The proposer-slice operator on sets of Proposal-tuples is defined as:

$\pi^{PV\text{-}atv}_{p}(s)=\{ \pi^{PV\text{-}atv}_{p}(up) : up\in s\}$.

Transaction and valid timeslice operators $\rho^{PV\text{-}atv}_{T1}(r)$ and $\tau^{PV\text{-}atv}_{T2}(r)$ on sets of update-proposals are defined similarly. The slice operators can be straightforwardly adapted to operate on proposals in which one or more dimensions (proposer, transaction time, valid time) have been removed by a slicing operator. For instance, $\tau^{PV\text{-}tv}_{T2}$ is the valid-timeslice operator on Proposal-tuples in which only transaction and valid times are present.

**Definition 3.3.2: Snapshot equivalence on sets of Update-proposals**. Two sets of Update-proposals r and s are

snapshot equivalent if for all the transaction times T1 not exceeding the current time, for all the valid times T2 and for all proposers p:

$\tau^{PV\text{-}t}{}_{T2}(\rho^{PV\text{-}tv}{}_{T1}(\pi^{PV\text{-}atv}{}_{p}(r))) = \tau^{PV\text{-}t}{}_{T2}(\rho^{PV\text{-}tv}{}_{T1}(\pi^{PV\text{-}atv}{}_{p}(s)))$. ♦

Given the above definitions, we can prove that Property 3.3.3 holds. It is worth stressing that Conditions 3.2.3.2 and 3.2.3.5 are essential to obtain such a fundamental property, which "certifies" the semantic clarity of the data model we use.

**Property 3.3.3: Uniqueness of model on sets of Update-proposals.** Two sets of Update-proposals defined over the same type are snapshot equivalent if and only if they are identical. ♦

Analogously, slicing operators and snapshot equivalence can be defined for evaluator relations, sets of proposals of insertion, and sets of proposals of deletion. We have proved that Property 3.3.4 holds as well:

**Property 3.3.4:** In our data model, identity and snapshot equivalence coincide, i.e., two databases over the same evaluator schema in our model are identical if and only if the corresponding evaluator relations and sets of proposals of insertion, proposals of deletion and Update-proposals are snapshot equivalent. ♦

Reducibility to BCDM is also a key result:

**Property 3.3.5: Reducibility of BCDM$^{PV}$ data model to BCDM data model**. The BCDM$^{PV}$ data model reduces to the BCDM data model in case no proposals are proposed/evaluated. ♦

The property of reducibility to BCDM holds, since the pair <DB_Evaluators, DB_Proposers> trivially reduces to a BCDM database in case only one level of data (i.e., DB_Evaluators) is taken into account, and evaluators' information is disregarded. This case models the "non-proposal vetting" context in which users can directly operate insert/delete/update operations on the data and no evaluation is needed.

## 4 Manipulation Operations

In this section we define the manipulation operations of BCDM$^{PV}$. We introduce two levels of operations: proposer operations and evaluator operations. As regards proposer operations, we define *proposal of insertion*, *proposal of deletion* and *proposal of update*. On the other hand, evaluators can either *accept* or *reject* proposals. Our choice of defining an independent operator for proposals of update is a departure from the relational tradition, in which updates are usually implemented by transactions containing a deletion and an insertion. However, since one of the core features of our approach is that of modeling proposals of update as primitive concepts (see the discussion in Section 3.2.3), defining primitive operations applied directly on them was the most natural and effective choice. Actually, as we will see later on, in the proposal vetting context, the acceptance of a proposal of update does not trivially correspond to the acceptance of a deletion and of an insertion. As a matter of fact, (i) a proposal of update may concern a proposal of insertion, so that its acceptance does not require the deletion of any tuple in the evaluators database; (ii) in all cases, the acceptance of an update proposal involves the rejection of all its alternatives (which would not be involved by the acceptance of a deletion followed by an insertion).

### 4.1 Proposer operations

In the following we present the definition of the operation of propose_update, which is the most complex one. Given a relation r∈DB_Evaluators, a proposal of update can be used in order to modify (i) a tuple in r, or (ii) a tuple in pi(r), or (iii) an alternative of an Update-proposal in pu(r) (we thus allow chaining of update proposals, to support incremental updates, i.e., further updates to an already existing proposal of update). As explained in the previous sections, proposal operations are stored in DB_Proposals waiting for an acceptance or a rejection. Specifically, in all cases (i)-(iii) above, the result of a proposal of update is an Update-proposal which, depending on the cases, may be a newly generated one or a modification of an already existing Update-proposal. The definition of *propose_update* is quite complex, since it has to cover the three cases, granting also that the operation is admissible (e.g., that it refers to existing tuples) and that it does not introduce incorrect Update-proposals (e.g., value-equivalent origins, or value-equivalent alternatives to the same origin, see Conditions 3.2.3.2 and 3.2.3.5).

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n \mid T_e)$, the arguments of a *propose_update* operation regarding a tuple in r are: (a) *r* itself, (b) the old tuple to be modified, and (c) the new tuple (i.e., $(a_1'',\ldots,a_n'' \mid p_{new},t_{vt\_new})$). While the tuple in (c) always has the schema $(A_1,\ldots,A_n \mid Proposers \times D_{VT})$, we specify the old tuple in different ways, depending on the case (i)-(iii) we cope with. Specifically, if we have to cope with an update to an alternative of an Update-proposal in pu(r), the alternative is uniquely identified by a pair <origin,alternative> (i.e., $<(a_1,\ldots,a_n),(a_1',\ldots,a_n')>$).[6] On the other hand, if we cope with an update to an evaluator tuple or to a proposal of insertion, the old tuple is uniquely identified by its explicit values (i.e., $(a_1,\ldots,a_n)$). In order to deal with this case within the above pattern, in such cases we assume that the old tuple is specified by the pair $<(a_1,\ldots,a_n),(a_1,\ldots,a_n)>$.

The propose_update operation first checks the applicability of the proposal operation, through the admissible_propose_update routine, which also checks the temporal consistency of data.

**Definition 4.1.1: admissible_propose_update**. Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n \mid T_e)$, let A stand for $(A_1,\ldots,A_n)$, and let $<(A_1,\ldots,A_n), (A_1,\ldots,A_n \mid T_p)>$ be the type of pu(r). We define admissible_propose_update, applied to an operation propose_update(r,$<(a_1,\ldots,a_n),(a_1',\ldots,a_n')>$,$(a_1'',\ldots,a_n'' \mid p_{new}, t_{vt\_new})$), as follows:

admissible_propose_update( propose_update(r,$<(a_1,\ldots,a_n)$, $(a_1',\ldots,a_n')>$, $(a_1'',\ldots,a_n'' \mid p_{new},t_{vt\_new})$) ):

[6] Since in our model we may have value-equivalent alternatives which belong to different Update-proposals, a given alternative can be uniquely identified only if also the Update-proposal it belongs to is specified. Since each Update-proposal in a set of Update-proposals is uniquely identified by its origin, we specify a given alternative through the pair <origin, alternative>. Finally, notice that, since we disallow value-equivalent alternatives to the same origin (see Condition 3.2.3.2), the implicit attributes are not needed to identify it.

(1) ($\exists$x$\in$r : (x[A]=($a_1$,...,$a_n$) $\wedge$ current(x)) $\vee$ $\exists$x$\in$pi(r) : (x[A]=($a_1$,...,$a_n$) $\wedge$ current(x))) $\wedge$
(2) ($\exists$up$\in$pu(r) : (origin(up)=($a_1$,...,$a_n$) $\wedge$ $\exists$ y $\in$ alternatives(up) : (y[A] = ($a_1$',...,$a_n$') $\wedge$ current(y)) $\vee$ ($a_1$,...,$a_n$)= ($a_1$',...,$a_n$')) ) $\wedge$
(3) $\forall$k$\in$ r ((k[A]= ($a_1$'',...,$a_n$'') $\wedge$ current(k)) $\Rightarrow$ ($a_1$'',...,$a_n$'')=($a_1$,...,$a_n$)) $\wedge$
(4) $p_{new} \in$ Proposers♦

A proposal of update is admissible if a conjunction of four conditions (above tagged as (1)-(4)) holds:

(1) *($a_1$,...,$a_n$)* identifies a tuple *x* in the evaluator relation *r* or in the proposal of insertion set *pi(r)* and such a tuple is current;
(2) either (i) the input *<($a_1$,...,$a_n$),($a_1$',...,$a_n$')>* identifies a current alternative of an Update-proposal *up* in *pu(r)*, or (ii) it identifies a tuple in *r* or (iii) in *pi(r)* (given the convention on the input format we have discussed above, the condition *($a_1$,...,$a_n$)*=($a_1$',...,$a_n$') holds exactly in cases (ii) and (iii));
(3) there is no current tuple *k*$\in$*r* which is value equivalent to the new proposal *($a_1$'',...,$a_n$'')*, except (possibly) the origin itself. In such a case, the proposal concerns an update to the valid time of the origin. This condition is used to disallow a new proposal *($a_1$'',...,$a_n$'' | $p_{new}$, $t_{vt\_new}$)* which is value equivalent to a current tuple *t'* in the evaluator level relation, which is not the origin of the Update-proposal to be modified. In fact, if accepted, the new proposal would be value equivalent to *t'*, and value-equivalent tuples are not admitted in *r*;
(4) the proposer $p_{new}$ belongs to the set of proposers.

In order to simplify the definition of propose_update, we introduce a function to create Update-proposals.

**Definition 4.1.2: create_up**. Given a relation in DB_Evaluators with schema R=($A_1$,…,$A_n$ | $T_e$), create_up takes as an input an origin o defined on the schema ($A_1$,…,$A_n$) and a set {$alt_1$,…,$alt_m$} of (non-value-equivalent) alternatives on the schema ($A_1$,…,$A_n$ | $T_p$), and gives as an output an Update-proposal of type <($A_1$, …,$A_n$), ($A_1$, …,$A_n$ | $T_p$)> having o as an origin and {$alt_1$,…,$alt_m$} as alternatives, i.e.,

create_up(o,{$alt_1$,...,$alt_m$}) =<o,Alt($alt_1$,...,$alt_m$)>. ♦

We can now define our operator to propose updates.

Note that the function *current* returns *true* if the tuple has the UC value in its transaction time (i.e., it is present at the current time).

In the formula, we assume the standard "nesting" policy for the scope of the variables in the conditions.

**Definition 4.1.3: propose_update.** Given a relation r$\in$DB_Evaluators with schema R=($A_1$,…,$A_n$ | $T_e$), let A stand for ($A_1$,…,$A_n$), let <($A_1$, …,$A_n$), ($A_1$, …,$A_n$ | $T_p$)> be the type of pu(r). We define propose_update as follows:

propose_update(r, <($a_1$,...,$a_n$), ($a_1$',...,$a_n$')>, ($a_1$",...,$a_n$" | $p_{new}$, $t_{vt\_new}$)):
**if**(admissible_propose_update(propose_update(r,<($a_1$,...,$a_n$), ($a_1$',...,$a_n$')>,($a_1$'',...,$a_n$''| $p_{new}$, $t_{vt\_new}$)))) **then**
**begin**
(1) **if** ($\neg\exists$ up $\in$ pu(r) : origin(up)=($a_1$,...,$a_n$)) **then**
pu(r)$\leftarrow$pu(r)$\cup${create_up(($a_1$,...,$a_n$),{($a''_1$,...,$a''_n$ | {$p_{new}$}×{UC}×$t_{vt\_new}$)})}
(2) **else if** ($\exists$ up $\in$ pu(r) : (origin(up)=($a_1$,...,$a_n$) $\wedge$ $\forall$y $\in$alternatives(up) y[A]$\neq$($a_1$'',...,$a_n$'') )) **then**
pu(r)$\leftarrow$pu(r) –{up} $\cup$ {create_up(($a_1$,...,$a_n$), alternatives(up) $\cup$ {($a_1$",...,$a_n$"| {$p_{new}$}×{UC}×$t_{vt\_new}$)})}
(3) **else if** ($\exists$ up $\in$ pu(r) : (origin(up)=($a_1$,...,$a_n$) $\wedge$ $\exists$ y $\in$ alternatives(up) : y[A] = ($a_1$'',...,$a_n$'') $\wedge$ $p_{new}$ $\notin$y[Proposer])) **then**
pu(r)$\leftarrow$pu(r)–{up}$\cup${create_up(($a_1$,...,$a_n$), alternatives(up) – y $\cup$ {($a_1$'',...$a_n$''|y[$T_p$] $\cup$ {$p_{new}$}×{UC}×$t_{vt\_new}$)})}
(4) **else if** ($\exists$ up $\in$ pu(r) : (origin(up)=($a_1$,...,$a_n$) $\wedge$ $\exists$ y $\in$ alternatives(up) : y[A] = ($a_1$'',...,$a_n$'') $\wedge$ $p_{new} \in$ y[Proposer])) **then**
pu(r)$\leftarrow$pu(r)–{up}$\cup${create_up(($a_1$,...,$a_n$), alternatives(up) – y $\cup$ {($a_1$'',...$a_n$''|y[$T_p$] – {$p_{new}$}× uc_ts($\pi^{atv}_{p_{new}}$(y)[T]) $\cup$ {$p_{new}$}×{UC}×$t_{vt\_new}$)})}
**end**♦

First, the admissibility of the update proposal is checked. If admissibility holds, four different cases must be considered (otherwise the operation has no effect, and an appropriate warning may be signaled):

(1) the input origin *($a_1$,...,$a_n$)* does not identify any already existing Update-proposal *up*$\in$*pu(r)*; in such a case a new Update-proposal is inserted into *pu(r)*, having as an origin the input origin *($a_1$,...,$a_n$)* and having the input proposal as the (only) alternative. Notice that the new triple $T_p$ of the alternative has the input $p_{new}$ as a proposer, UC as a transaction time, and the input $t_{vt\_new}$ as a valid time (i.e., it is obtained by performing the Cartesian product {$p_{new}$}×{UC}×$t_{vt\_new}$);
(2) the input *($a_1$,...,$a_n$)* identifies an already existing Update-proposal *up*$\in$*pu(r)*, and there is not any alternative proposal in *up* which is value equivalent to the current proposal *($a_1$'',...,$a_n$'')*. In such a case the new alternative *($a_1$'',...,$a_n$'' | {$p_{new}$}×{UC}×$t_{vt\_new}$)* is added to the already existing alternatives of *up*;
(3) the input *($a_1$,...,$a_n$)* identifies an already existing Update-proposal *up*$\in$*pu(r)*, the input *($a_1$'',...,$a_n$'')* identifies an existing alternative proposal in *up*, but such an alternative has not been proposed by the proposer $p_{new}$. In such a case the alternative is updated with a new triple, which contains the new temporal information and the new proposer (i.e., {$p_{new}$}×{UC}×$t_{vt\_new}$). Notice that adding *($a_1$'',...,$a_n$'')* as a new alternative is not possible, since value-equivalent alternatives are not admitted (see Condition 3.2.3.2);
(4) the input *($a_1$,...,$a_n$)* identifies an already existing Update-proposal *up*$\in$*pu(r)* and the proposer $p_{new}$ has already proposed an alternative of *up* value equivalent to *($a_1$'',...,$a_n$'')*. In such a case, $p_{new}$ proposes to change the valid time associated with the tuple. Thus, (1) the old triples having $p_{new}$ as a proposer must be made not current (henceforth: must be "closed"), and (2) all the triples containing the new bitemporals must be inserted – as in case (3). The closure is obtained by removing all the old triples whose proposer is $p_{new}$ and whose transaction time is UC: $\pi^{atv}$ selects all the tuples whose proposer is $p_{new}$, and *uc_ts* is a function that gives as an output the set of all bitemporal chronons (UC, $c_v$) (i.e., all chronons having UC as their transaction time) from the bitemporal timestamp of the tuple, defined as in BCDM [2,8].

**Example.** The proposal of update issued by proposer $p_2$ at step 6 is coped with in our approach as propose_update(INDEP, <(Poland, dictatorship), (Poland, dictatorship)>, (Poland, monarchy | $p_2$, [1025,1595])). In

particular, proposer $p_2$ issues a proposal of update to the proposal issued by $p_1$ at step 3, which refers to the tuple (Poland, dictatorship) in the DB_Evaluators relation INDEP. The proposal of update is admissible and branch (2) in formula 4.1.3 is then taken. Fig. 2(G) shows the effect of such an operation, at transaction time 7 (see the new alternative of the origin (Poland, dictatorship)). ♦

It is worth noticing that, although we cope with chains of proposals of update, we do not explicitly store the whole chaining of updates, since we directly relate each proposal to the original tuple to be modified (and not to the alternative proposal it directly modifies). This is a deliberate choice we made to simplify both the data model and the definition of the algebraic operations.

### 4.2 Evaluator operations

In our approach, evaluators can reject or accept proposals in DB_Proposers. Since we want to retain the whole database history, a rejected proposal is not physically deleted from DB_Proposers; instead, it is made not current by "*closing*" its implicit attributes: i.e., the triples with UC as a transaction time have to be removed (compare the transaction times [5,UC] and [5,5] of the second alternative of (Poland, dictatorship) in Fig. 2(E) and in Fig. 2(F), representing the rejection of the alternative itself at step 5). On the other hand, the acceptance of a proposal is used by evaluators to make a given current proposal effective, i.e., to execute it on the DB_Evaluators relation. Notice that, besides causing a modification of DB_Evaluators, the acceptance of a proposal could also have some effects on DB_Proposers, since proposals that are alternatives of the accepted one need to be "closed".

Now we define the operation *accept_update* of acceptance of a proposal of update. *Accept_update* is used by evaluators to update DB_Evaluators according to the given proposal. Only proposals that are current may be accepted by evaluators. As anticipated, the acceptance of a proposal of update must "close" the alternative proposals. Moreover, the accepted tuple as well as the deletion and/or insertion proposals concerning the tuple itself must be "closed". Such operations are performed through the *delete_alternatives* routine.

The arguments of the *accept_update* operation are the DB_Evaluators relation *r* to be modified, the explicit part of the selected alternative $<(a_1,...,a_n),(a_1',...,a_n')>$ of an Update-proposal in *pu(r)*, the selected valid time $t_{vt}$ (notice that different proposers may have proposed different valid times for the same explicit part of the proposal), and the evaluator *e*.

As a first step, admissible_accept_update is invoked in order to check the acceptability of the operation, considering also the temporal consistency of data. Notice that, since the data stored in the database could change, it is possible that a proposal is admissible (i.e., it is consistent with the status of DB_Evaluators) when it is issued, but it is no longer admissible at acceptance time. Thus some checks have to be repeated at acceptance time.

**Definition 4.2.1: admissible_accept_update.** Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n \mid T_e)$, let $<(A_1, \ldots,A_n), (A_1, \ldots,A_n \mid T_p)>$ be the type of pu(r); we define admissible_accept_update as follows (let A stand for $A_1,\ldots,A_n$):

admissible_accept_update( accept_update(r, $<(a_1,...,a_n)$, $(a_1',...,a_n')>$, e, $t_{vt}$) ):
(1) ∃ up∈pu(r) : origin(up)=$(a_1,...,a_n)$ ∧
(2) ∃y ∈ alternatives(up) : y[A] = $(a_1',...,a_n')$ ∧ ∃ p : {UC}×$t_{vt}$ = uc_ts($\pi^{atv}_p$(y)[T]) ∧
(3) ∀ z∈r (z[A]=$(a_1',...,a_n')$ ∧ current(z) ⇒ $(a_1',...,a_n')$ =$(a_1,...,a_n)$) ∧
(4) e∈Evaluators♦

The operation is admissible if the conjunction of four conditions holds:
(1) the input origin $(a_1,...,a_n)$ identifies an Update-proposal $up \in pu(r)$;
(2) $(a_1',\ldots,a_n')$ identifies a current alternative of *up*, having $t_{vt}$ has its valid time;
(3) there is no current tuple $z \in r$ which is value equivalent to the chosen alternative $(a_1',...,a_n')$, except (possibly) the origin itself (see the comments to Definition 4.1.1, part (3));
(4) *e* is an evaluator.

We can finally define the accept_update operation.

**Definition 4.2.2: accept_update.** Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n \mid T_e)$, let $<(A_1, \ldots,A_n), (A_1, \ldots,A_n \mid T_p)>$ be the type of pu(r); we define accept_update as follows (let A stand for $A_1,\ldots,A_n$):

accept_update(r, $<(a_1,...,a_n)$, $(a_1',...,a_n')>$, e, $t_{vt}$)
**if** (admissible_accept_update(accept_update(r, $<(a_1,...,a_n)$, $(a_1',...,a_n')>$, e, $t_{vt}$))) **then**
**begin**
(1) **if** (¬∃x∈r : x[A]=$(a_1,...,a_n)$ ∧ current(x)) ∧ ( ∃y∈pi(r) : y[A] = $(a_1,...,a_n)$ ∧ current(y)) **then**
insert$^{PV}$(r,$(a_1',...,a_n')$, e, $t_{vt}$);
delete_alternatives(r, $(a_1,...,a_n)$)
(2)**else if** ( ∃x∈r: x[A]=$(a_1,...,a_n)$ ∧ current(x) ) **then**
delete$^{PV}$(r, $(a_1,...,a_n)$);
insert$^{PV}$ (r,$(a_1',...,a_n')$, e, $t_{vt}$);
delete_alternatives(r, $(a_1,...,a_n)$)
**end**♦

where delete$^{PV}$ and insert$^{PV}$ are straightforward adaptations of the BCDM operations of insertion and deletion to cope also with evaluators.

If the operation is admissible, two cases must be distinguished:
(1) the evaluator is accepting an update to a proposal of insertion. In such a case, the *accept_update* operation inserts into *r* the new tuple and "closes" the Update-proposal *up* and (possibly) proposals of insertion and deletion concerning $(a_1,...,a_n)$ (through *delete_alternatives*). Since the update concerns a new tuple proposed for insertion, in this case, no tuple needs to be deleted from the evaluator relation *r*;
(2) the evaluator is accepting an update to a tuple in *r*. In such a case, the *accept_update* operation first deletes the tuple $(a_1,...,a_n)$ from *r*, and then performs the same operations as in case 1 above.

**Example.** Referring to our running example, at step 9 $e_1$ accepts the proposal of update issued by proposer $p_2$ at step 6 through an operation accept_update(INDEP, <(Poland, dictatorship), (Poland, monarchy)>, $e_1$, [1025,1595]). The accept operation is admissible and the branch (2) in formula 4.2.2 is taken. The *accept_update* routine first logically deletes (using delete$^{PV}$) the tuple (Poland, dictator-

ship) from *INDEP* and then calls insert$^{PV}$ to insert into *INDEP* the chosen proposal. Moreover, *accept_update* also executes the *delete_alternatives* routine to "close" such an Update-proposal. Fig. 2(I) reports the resulting Update-proposal in *pu(INDEP)* and the updated content of the relation *INDEP* after the acceptance at step 9. ♦

### 4.3 Properties of manipulation operations

In most cases (consider, e.g., TSQL2) temporal approaches have been devised in such a way to be a consistent extension of conventional ones [8], in order to guarantee interoperability with pre-existent approaches. Since our approach extends BCDM, we might aim at providing a consistent extension of BCDM manipulation operators. Unfortunately, such a property cannot hold for our approach: since in the proposal vetting context direct insertion/deletion/update operations are not supported[7], BCDM$^{PV}$ manipulations operations cannot be a consistent extensions of BCDM.

Nevertheless, we devised our approach in such a way that the following (less strict) property holds:

**Property 4.3.1: "Proposal vetting" consistent extension of BCDM.** If all users are both evaluators and proposers, our model is a *"proposal vetting" consistent extension* of the BCDM model (considering data in DB_Evaluators, and neglecting the "Evaluator" implicit attribute), since each manipulation operation Op$^{B}$ in BCDM can be performed as a pair of operations <propose_Op; accept_Op> in our approach, leading to the same results, for the data in DB_Evaluators only. ♦

## 5 Relational Algebra

Besides manipulation operation, also query operators must be provided, in order to support the possibility of querying data, selecting and joining them. They can help evaluators in taking their acceptance/rejection decisions, as well as proposers in proposing updates to data. For instance, in step 8 in the example, an evaluator requires joining the INDEP and CAPITAL tables in order to "reconstruct" proposals concerning the independence of Poland when its capital was Cracow. Additionally, since the evaluator requires that only current proposals are taken into account, also a form of temporal selection on the transaction time is involved in the query at step 8.

Since in this paper we operate at the semantic level, the query language for our extended data model is provided at the algebraic level, as an extension of BCDM temporal algebra[8].

We have extended the BCDM model by including the evaluator (or the proposer) as an implicit attribute. We therefore extend the BCDM algebraic operators to cope with the new implicit attribute. As an example, we propose the natural join operator on DB_Evaluators relations. The other basic Codd's operators (union, difference, selection and projection) can be defined in a similar way.

**Definition 5.1: natural join $\bowtie^{E}$.** Given two relations $r_1 \in$DB_Evaluators and $r_2 \in$DB_Evaluators with schema $R_1=(A_1,\ldots,A_n, B_1,\ldots,B_m \mid T_e)$ and $R_2=(A_1,\ldots,A_n, C_1,\ldots,C_k \mid T_e)$ respectively, natural join $\bowtie^{E}$ provides as an output a relation over the schema $(A_1,\ldots,A_n, B_1,\ldots,B_m, C_1,\ldots,C_k \mid T_e)$ defined as follows (let A stand for $A_1,\ldots,A_n$, B for $B_1,\ldots,B_m$ and C for $C_1,\ldots,C_k$):

$r_1 \bowtie^{E} r_2 = \{ z : \exists t_1 \in r_1, \exists t_2 \in r_2 : z[A]=t_1[A]=t_2[A] \wedge z[B]=t_1[B] \wedge z[C]=t_2[C] \wedge z[T_e]=t_1[T_e] \cap t_2[T_e] \wedge z[T_e] \neq \emptyset \}$.♦

Reducibility is one of the most important properties in the temporal database area, to grant that the semantics of basic algebraic operators is preserved by the temporal extended operators [8,17]. In order to prove that our extended algebraic operators are reducible to BCDM ones, we used the evaluator-slice (and proposer-slice, which is analogous, see Section 3) operator:

$\pi^{E\text{-}atv_e}(r)=\{z : \exists x \in r : z[A]=x[A] \wedge z[T]=\{(t,v) : (e,t,v) \in x[T_e]\} \wedge z[T] \neq \emptyset\}$.

**Property 5.2: Reducibility.** Our algebra for DB_Evaluators, proposals of insertion, and proposals of deletion reduces to BCDM algebra, i.e., for each algebraic unary operator Op$^{E}$ in our model, and indicating with Op$^{B}$ the corresponding BCDM operator, for each relation r in DB_Evaluators, and for any evaluator e, the following holds (the analogous holds for binary operators):

$$\pi^{E\text{-}atv_e}(Op^{E}(r)) = Op^{B}(\pi^{E\text{-}atv_e}(r)).$$ ♦

Additionally, the treatment of proposals of update demands for the definition of new algebraic operators operating on sets of Update-proposals.

We present here the natural join operator on sets of Update-proposals. The other basic operators (union, difference, selection and projection) can be defined in a similar way. We characterize the output of natural join as a set of Update-proposals z of the general form <origin(z), alternatives(z)>, that can be defined by alternative cases. In the formula, we assume the standard "nesting" policy for the scope of the variables in the conditions.

**Definition 5.3: natural join $\bowtie^{PV}$.** Given the sets of Update-proposals $s_1=pu(r_1)$ and $s_2=pu(r_2)$ corresponding to relations $r_1 \in$DB_Evaluators and $r_2 \in$DB_Evaluators with schema $R_1=(A_1,\ldots,A_n, B_1,\ldots,B_m \mid T_e)$ and $R_2=(A_1,\ldots,A_n, C_1,\ldots,C_k \mid T_e)$ respectively, let the types of $s_1$ and $s_2$ be $<(A_1,\ldots,A_n,B_1,\ldots,B_m), (A_1,\ldots,A_n, B_1,\ldots,B_m \mid T_p)>$ and $<(A_1,\ldots, A_n, C_1, \ldots, C_k), (A_1,\ldots,A_n, C_1,\ldots,C_k \mid T_p)>$ respectively. Natural join $\bowtie^{PV}$ provides as an output a set of Update-proposals defined as follows (let A stand for $A_1,\ldots,A_n$, B for $B_1,\ldots,B_m$ and C for $C_1,\ldots,C_k$):

$s_1 \bowtie^{PV} s_2 = \{$ <origin(z),alternatives(z)> :
**if** $\exists up_1 \in s_1, \exists up_2 \in s_2 :$ origin($up_1$)[A]=origin($up_2$)[A] $\wedge$
$\exists alt_1 \in$ alternatives($up_1$), $\exists alt_2 \in$ alternatives($up_2$) :
$alt_1[A]=alt_2[A] \wedge alt_1[T_p] \cap alt_2[T_p] \neq \emptyset$ **then**
origin(z)[A]← origin($up_1$)[A]; origin(z)[B]← origin($up_1$)[B]; origin(z)[C]← origin($up_2$)[C];
alternatives(z) = {alt :
alt[A] ←$alt_1$[A]; alt[B] ←$alt_1$[B]; alt[C] ←$alt_2$[C];
alt[$T_p$] ←$alt_1[T_p] \cap alt_2[T_p]$}} ♦

[7] As a matter of fact, we could trivially extend our approach to let evaluators directly insert and delete tuples in relations in DB_Evaluators. With such an extension, our approach is trivially a consistent extension of BCDM, as regards manipulation operations.

[8] Notice that, in our approach, algebraic operators are used only to query data, since the tuples in the relations resulting from the application of algebraic operators cannot be directly accepted/rejected by evaluators. In fact, such operations would be meaningless, since accepted proposals must conform the schema of data at the evaluators' level.

The result of proposal-vetting natural join is a set of Update-proposals built as follows. Two Update-proposals $up_1$ and $up_2$ with origins value equivalent on the common attributes $A_1,\ldots,A_n$ are merged into one Update-proposal having as origin the standard natural join of the origins. The alternatives of the new tuple are built by performing the standard natural join on the explicit attributes and the intersection of the implicit attributes. Only if this intersection is not empty the alternative is stored as an output.

We have defined our algebraic operators on sets of Update-proposals in such a way that they have the property of reducibility [8] with respect to BCDM algebraic operators. Proposals of update cannot be directly modeled within BCDM, mainly due to the fact that (i) they cope with alternative pieces of information (while in BCDM only conjunctive information can be coped with), and (ii) they also model proposers (as an implicit attribute). Thus, reduction to BCDM involves, *for each* Update-proposal, the choice of *at most one* of its alternatives, and the choice of a proposer. After these choices, each resulting Update-proposal (having just one alternative) can be easily mapped onto BCDM, by converting it onto a tuple of a relation with the proper schema. The *alternative-slice* operator is introduced in order to select alternatives. Specifically, the reduction to at most one alternative for each Update-proposal is obtained by fixing a specific value for each one of the attributes. The proposer-slice operator $\pi^{PV\text{-}atv_p}$ is used for selecting the proposer.

**Note.** Here we follow the methodology usually adopted in order to reduce BCDM (and temporal approaches in general) to the standard relational model. In that case, a specific time is specified as a slicing criterion and the temporal dimension is removed. In our approach, a specific set of values for the attributes and one proposer are chosen, so that the "alternative" and the "proposer" dimensions are removed.

**Definition 5.4: Alternative-slice operator on sets of Update-proposals.** Given a set of Update-proposals $s$ defined over the type $<(A_1,\ldots,A_n), (A_1,\ldots,A_n \mid T)>$, the result of the alternative-slice operator $\mu_{a1',\ldots,an'}(s)$ is a BCDM relation defined over the schema $(A_1,\ldots,A_n, A_1',\ldots,A_n' \mid T)$ (where attributes $A_1',\ldots,A_n'$ are a renaming of $A_1,\ldots,A_n$ respectively) defined as follows:

$\mu_{a1',\ldots,an'}(s)=\{(a_1,\ldots,a_n,a_1',\ldots,a_n' \mid t) :$
$\exists up \in s : (a_1,\ldots,a_n)=origin(up) \wedge$
$(a_1',\ldots,a_n' \mid t) \in alternatives(up)\}$ ♦

**Property 5.5: Reducibility of BCDM$^{PV}$ algebra on sets of Update-proposals to BCDM algebra.** BCDM$^{PV}$ algebraic operators on sets of Update-proposals are reducible to BCDM algebraic operators, i.e., for each algebraic unary operator $Op^{PV}$ in our model, and indicating with $Op^B$ the corresponding BCDM operator, for each set of Update-proposals $s$, the following holds (the analogous holds for binary operators):

$$\mu_{a1',\ldots,an'}(\pi^{PV\text{-}atv_p}(Op^{PV}(s))) = Op^B(\mu_{a1',\ldots,an'}(\pi^{PV\text{-}atv_p}(s))),$$

where $a_1',\ldots,a_n'$ are arbitrary values in the domains of the respective attributes and p is a proposer. ♦

Finally, given the fact BCDM algebraic operators reduce to relational algebra operators [8], also Corollary 5.6 trivially holds.

**Corollary 5.6: Reducibility of BCDM$^{PV}$ algebra to relational algebra.** The BCDM$^{PV}$ algebraic operators are reducible to relational algebra operators. ♦

For the sake of brevity, we do not report the exhaustive listing of all our extended algebraic operators. However, it is worth mentioning that in our extended algebra we also provide: (i) extended versions of algebraic operators to cope with "mixed" cases in which sets/relations have different types (e.g., natural join between proposals of update having different implicit attributes or between proposal of updates and Evaluator relations); (ii) slicing operators (e.g., $\pi^{PV\text{-}atv_p}$); (iii) reduction operators, that remove one of the implicit attributes; (iv) temporal selection operators ($\sigma^T$).

**Example.** The query at Step 8 in the Example can be expressed as follows:

$\sigma^T_{TT=UC}(\sigma_{Nation='Poland' \wedge City='Cracow'}(INDEP \bowtie^{PV} CAPITAL))$

where $\bowtie^{PV}$ is the natural join between two sets of proposal tuples, $\sigma$ is the standard selection on non-temporal attributes, and $\sigma^T$ is the temporal selection operator, used to select only current tuples. ♦

## 6 Implementation and Extensions

### 6.1 Implementation

As a proof of concept we have developed a prototypical implementation [24] of our approach on top of TIMEDB [19], a TSQL2-like system based on BCDM semantics. We have implemented a simple version of the data model described in this paper, in which only transaction time is considered, and manipulation operations are provided. (The realization of algebraic operators in such a tool is one of the goals of our future work.) Our implementation has been based on the general architecture for temporal DBs described in [8, Chapter 24]. In particular, an additional layer has been added on top TIMEDB to support proposal vetting. The parser has been extended to cope with the extended syntax. Thanks to the reducibility and consistent extension properties, in case no proposal vetting facility is used, standard TIMEDB operations are provided. On the other hand, proposal-vetting operations are implemented by taking advantage as much as possible of the operations provided by TIMEDB. The interpretation of DDL commands has been extended to associate with each relation three additional relations, implementing the set of proposals of insertion, deletions and updates. Such additional relations are stored as TIMEDB temporal relations, and are managed by the additional layer. Proposal and evaluation operations are defined in the additional layer, and operate on such relations (as well as on the reference relations, in the case of acceptance). The additional computational complexity of our implementation (with respect to TIMEDB) is quite limited. Besides an extension to the parsing, the additional cost is due to the need to store and manipulate the additional relations for proposals (which is a necessary cost for any approach to proposal vetting, which involves storing proposals waiting for an evaluation). Manipulation operations involve the admissibility checks, which require a check of the proper relations. Notice, however,

that admissibility checks are also required by TIMEDB (e.g., to check for the existence of the tuple to be modified). On the other hand, the acceptance of a proposal involves a significant extra-workload, since the "delete_alternatives" function is required to check the three proposal relations to "close" the alternatives of the accepted proposal. However, this cost is due to the intrinsic semantics of the operation of acceptance, in the context of mutually exclusive alternatives.

### 6.2 Granularity of operations

In this paper, we aim at proposing an approach which is *reducible* to BCDM. Thus, we have defined the manipulation and algebraic operations at the same granularity used in BCDM, i.e., we consider tuples as the primitive entities. However, the granularity at which proposals can be issued and evaluated may be either finer or coarser.

As an example of a finer granularity, a proposal could be accepted by evaluators only as concerns a part of its temporal extent. From the technical point of view, this possibility can be accomplished by simply modifying the admissible_accept_update definition (Definition 4.2.1) in this way: in line (2) $\exists$ p : {UC}×$t_{vt}$ = uc_ts($\pi^{atv}_{p}$(y)[T]) should be replaced by $\exists$ p : {UC}×$t_{vt}$ $\subseteq$ uc_ts($\pi^{atv}_{p}$(y)[T]) (i.e., $\subseteq$ instead of =) to state that a subset of the temporal extent of a proposal can be accepted.

Our approach can also be extended to cope with operations at a coarser level (i.e., proposals and evaluations taking into account sets of tuples) in at least two different ways. The simplest way is to import in our approach the standard notion of "transaction" of DBMS, so that proposers can enclose a sequence of proposals in a transaction, and, similarly, evaluators can enclose a sequence of acceptances/rejections. This extension is trivial, since the standard transaction mechanism can be used, and each enclosed proposal/evaluation operation retains exactly the semantics discussed in Section 4. Additionally, we might also support proposers with the possibility of declaring a sequence of proposals as a unique "macro-proposal", to be evaluated (either accepted or rejected) as an atomic piece of information by evaluators. The treatment of macro-proposals can be achieved by extending our data model with additional (system-managed) attributes to store, for each enclosed proposal, its "macro-proposal" identifier and its order in the sequence. Evaluation operations operate on "macro-proposals". (For the sake of generality, "atomic" proposals can be treated as "macro-proposals" containing just a proposal). Notably, a "macro-proposal" is simply interpreted as the sequential execution of the proposals constituting it, where each constituting proposal retains the basic semantics presented in Section 4. Transactions can also be used, to grant that both "macro-proposals" and their evaluations are treated as atomic operations by the DBMS.

### 6.3 Multiple levels of proposers/evaluators

Our model can be extended to deal with more than two levels of users. For example, in CASE approaches such as [25] three (and more) levels are supported: the level of developers, the level of integrators of modules and the level of supervisors for releasing a final version. Of course, different policies can be supported, involving different extensions to our basic approach. One possible policy enforces a rigid ordering of levels, so that, to become effective, a proposal issued at level *i* must be approved, in the ordering of levels, by all higher levels (e.g., developers' proposals must be accepted by integrators first, and then by supervisors). Such a policy can be achieved as a generalization of our approach as discussed below. *n* different levels of users have to be defined. *Level-1* users can only propose operations, while *level-i* users can evaluate proposals from level *i-1* (and possibly issue new proposals). Rejections "close" proposals, while acceptances "propagate upward" proposals. Thus, the semantics of an acceptance by a user *u* at level *i* of a proposal *p* issued by users $\{u_1,\ldots,u_k\}$ involves the proposal of *p* by users $\{u_1,\ldots,u_k,u\}$ at level *i+1* (except in the case *i=n*, for which the semantics of our evaluators operations is maintained unchanged). From a semantic (abstract) point of view, such an extension would require, for each relation, a set of proposals of insertion, deletion and update at each level (reference relations correspond to the top level). This strategy enables the realization of the extension without major qualitative changes to the data model. However, the extension to multiple levels clearly implies substantial changes to the "process-level" description (i.e., the Petri Net, in which additional places and transitions would be required to capture the behavior of the additional user levels.)

## 7 Related Works

Computer Supported Cooperative Work (CSCW) is a widely spread paradigm. In the CSCW time/space matrix [26], proposal vetting would be classified in the class "different time / different place" of interactions, meaning that users can interact asynchronously, being in different physical locations. Such a type of interaction, finalized to the cooperative modeling / update of shared data/knowledge, is an important paradigm in Computer Science, and becomes more and more important and spread due to the large-scale availability of the Internet.

Some ad-hoc implementations have been built in order to cope with the proposal vetting phenomenon (consider, e.g., Citizendium). Additionally, mostly in the Object Oriented DB context, several general approaches have been developed in order to cope with data versioning and with some of the issues related to proposal vetting (see, e.g., the survey in [27]). Recently, some *object-oriented* approaches also consider, besides data versioning, valid and transaction times (see, e.g., [28-30]). A main difference between object-oriented approaches and relational approaches has been pointed-out by Sciore [31, page 425]: "*The relational model has a limited modeling capacity, and so researchers in historical relations have all being forced to extend the relational model in some way. On the other hand, object-oriented models are able to encapsulate the notion of time in classes. Thus there is no need to develop a new historical object-oriented model; what we need is a methodology for using these classes in our existing model*".

A few works have taken into account some form of support for alternatives within the relational context. For instance, Sarda and Reddy's work [32] allows one to represent events and actions in relational databases. It relies on the notion of "branching chronons", which represent transaction and valid times, associated with a propositional formula representing the possible occurrence of events. Although such an approach supports branching time and the possible evolutions of events, it does not provide direct support for proposal vetting.

The area of research on probabilistic databases [33] is (loosely) related to our work. A probabilistic database is an uncertain database in which possible, alternative worlds are modeled, each one with an associated probability. However, neither alternative world evolution, nor the conditions for selecting one specific alternative (as in [33]) are directly captured in this framework.

The proposal-vetting process is a kind of workflow, as shown in Section 2.1: in this direction, workflow approaches are related to our contribution. In this research area, attention has been mostly devoted to control flow, ignoring other perspectives, such as the data-related aspects of a workflow execution (while, as already discussed in Section 2.1, the treatment of data is indeed our main focus). In fact, "*data-centric workflow systems are currently a research area that deserves more efforts*" [34]. A few contributions to data-centric workflows can be cited. The work in [35] clarifies the interactions between control flow and data flow features in business process models, showing that ignoring data flow features can limit the flexibility of business process modeling languages. The work in [36] deals with workflow verification. While most analysis techniques typically abstract from data and check for errors such as deadlocks, livelocks, etc., this work looks for data-flow errors, such as accessing a data element that is not yet available. Consistently with our choice (see Section 2.1), in [36] data-flow representation relies on Petri Nets and their extensions (see also [37]). However, workflow verification is far from the scope of our research. The work in [38], deals with the problem of interaction between workflow instances, to support communication and collaboration. It highlights the need for smaller interconnected workflows starting from a data model (based on Petri Nets). However this contribution mainly concerns the composition of the functional part of the workflow, while the data necessary to actually execute it are considered secondary. The system in [39] tries to overcome this gap, and explicitly represents data (needed for composition). The workflow is modeled as a Petri Net, and a semantic description of data is provided by means of ontologies.

The research area of temporal workflow management systems is also (loosely) related to our work. In fact, temporal workflows can deal with the execution of different versions of a workflow schema (see e.g. [40], which deals with the representation of temporal clinical workflows). However, the contributions in this area typically do not pay attention to how versions can be built through a proposal vetting process.

To summarize, none of the above approaches provides specific support to identify the admissibility conditions for the manipulation operations, or to enforce the correct (i.e., consistent with the TDB theory) data manipulation semantics after proposal acceptance, or any support to a correct treatment of algebraic queries on the stored relational data. Thus BCDM$^{PV}$ is the only approach in the literature which directly copes with the proposal vetting phenomena at the semantic (in the sense of BCDM [8]; see footnote 4) level, and in the purely *relational* context. Indeed, since data-centric workflows are a rapidly evolving research area, we envision the possibility that such approaches can be adopted for developing an implementation of our semantic framework, at least as regards the proposal and evaluation operations. However, one of the main focuses of our approach concerns the definition of a temporal relational algebra for proposal vetting, which is, to the best of our knowledge, out of the scope of current research in data-centric workflows. For such a reason, the implementation we sketched in Section 6.1 is a more traditional DBMS-based one.

## 8 Conclusions

Proposal vetting is an emerging phenomenon, which often involves relational DBs (consider, e.g., Citizendium [3]). We propose BCDM$^{PV}$, a domain- and application-independent and theoretically grounded solution to proposal vetting in the relational context. In order to cope with the time of proposals/evaluations (transaction time), and, possibly, with the valid time of data, BCDM$^{PV}$ is grounded on the TDB theory, and, specifically, on the BCDM model. BCDM$^{PV}$ extends BCDM to support proposal vetting. In particular, the treatment of alternative proposals demands a major departure from the traditional relational model in general, and from BCDM in particular. We extended BCDM in such a way that (i) BCDM$^{PV}$ *data model* and (ii) BCDM$^{PV}$ algebra are reducible to the BCDM ones, and that (iii) BCDM$^{PV}$ *manipulation operations* are a *proposal vetting consistent extension* of BCDM ones. In such a way, we grant for the generality of our approach, for its implementability (on top of any TDB approach based on the BCDM semantics), and for the interoperability of such implementations with pre-existent TDBs and standard relational approaches.

## Acknowledgements

The authors are very indebted to R.T. Snodgrass, C. Dyreson, C. Combi, and W.M.P. van der Aalst for many useful comments and suggestions about a preliminary version of the work discussed in this paper. They are also very grateful to F. Grandi, S. Ram, and J. Roddick for their suggestions concerning references to related works.

## References

[1] http://www.wikipedia.org, Wikipedia, the free encyclopedia (URL last accessed on 12/07/2009).

[2] C.S. Jensen, R.T. Snodgrass, Semantics of Time-Varying Information, Information Systems, 21(4), 311–352, 1996.

[3] http://www.citizendium.org/, Citizendium, a citizens' compendium of everything (URL last accessed on 12/07/2009).
[4] P. Terenziani, S. Montani, A. Bottrighi, G. Molino, M. Torchio, Clinical guidelines adaptation: managing authoring and versioning issues, LNAI 3581, Springer-Verlag, Berlin, 151-155, 2005.
[5] L. Liu, M. Tamer Özsu (Eds.), Encyclopedia of Database Systems. Springer US, 2009.
[6] Y. Wu, S. Jajodia, X. Sean Wang: Temporal Database Bibliography Update. Temporal Databases, Dagstuhl: 338-366, 1997.
[7] R.T. Snodgrass, Developing Time-Oriented Database Applications in SQL, Morgan Kaufmann Publishers, Inc., San Francisco, July, 1999.
[8] R. T. Snodgrass (Ed.), The TSQL2 Temporal Query Language. Kluwer 1995.
[9] S.K. Gadia. A seamless generic extension of SQL for querying temporal data. Technical Report TR-92-02. CS Dept, Iowa State University, 1992.
[10] J. Ben-Zvi. The Time Relational Model. Ph.D. Dissertation, Computer Science Department, UCLA, 1982.
[11] L.E. McKenzie. An Algebraic Language for Query and Update of Temporal Databases. Ph.D. Dissertation, Computer Science Department, University of North Carolina at Chapel Hill, 1988.
[12] R.T. Snodgrass. The temporal query language TQuel. ACM Trans. on Database Systems, 12(2): 247-298, 1987.
[13] C.S. Jensen, L. Mark, N. Roussopoulos. Incremental implementation model for relational databases with transaction time. IEEE Trans. Knowl. Data Eng., 3(4): 461-473, 1991.
[14] P. Terenziani. Symbolic User-Defined Periodicity in Temporal Relational Databases. IEEE Trans. Knowl. Data Eng. 15(2), 489-509, 2003.
[15] P. Terenziani, R.T. Snodgrass. Reconciling Point-Based and Interval-Based Semantics in Temporal Relational Databases: A Treatment of the Telic/Atelic Distinction. IEEE Trans. Knowl. Data Eng. 16(5), 540-551 2004.
[16] L. Anselma, P. Terenziani, R.T. Snodgrass. Valid time indeterminacy in Temporal Relational Databases: A Family of Data Models. Proc. Temporal Representation and Reasonong (TIME) 2010, 139-145, 2010.
[17] M.H. Böhlen, C..S. Jensen, R.T. Snodgrass. Temporal Compatibility. pages 2936-2939 in [5].
[18] Oracle Database 10g Workspace Manager Overview. An Oracle White Paper http://www.oracle.com/technology/products/database/workspace_manager/pdf/twp_AppDev_Workspace_Manager_10gR2.pdf (URL last accessed on 05/07/2008).
[19] http://www.timeconsult.com/Software/Software.html (URL last accessed on 12/09/2009).
[20] C. Combi, G. Pozzi: Architectures for a temporal workflow management system. SAC 2004:659-666.
[21] J.L. Peterson. Petri Net Theory and the Modeling of Systems, Prentice Hall, 1981.
[22] G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad. Stochastic Well-formed Coloured nets and symmetric modeling applications. IEEE Transactions on Computers, 42(11), 1343 – 1360, 1993.
[23] K.R. Dittrich, R.A. Lorie, Version Support for Engineering Database Systems. IEEE Trans. Software Eng. 14(4): 429-437, 1988.
[24] A. Vigo. Estensioni alle basi di dati temporali per il supporto alla proposta ed alla valutazione di modifiche di dati, laura degree thesis in computer science, Università del Piemonte Orientale, 2009.
[25] SUN MICROSYSTEMS, Introduction to the NSE. SUN Part No. 800-2362-1300 (Mar. 7), 1988.
[26] R. Johansen. GroupWare: Computer Support for Business Teams. The Free Press, New York, NY, USA, 1988.
[27] R.H. Katz, Towards a Unified Framework for Version Modeling in Engineering Databases. ACM Comput. Surv., 22(4), 375-408, 1990.
[28] S. Gançarski, Database Versions to Represent Bitemporal Databases. In. LNCS, vol. 1677. Springer-Verlag, London, 832-841, 1999.
[29] M.M. Moro, N. Edelweiss, A.P. Zaupa and C.S. Santos, TVQL - Temporal Versioned Query Language. LNCS, vol. 2453. Springer-Verlag, London, 618-627, 2002.
[30] R. Machado, Á. F. Moreira, R. de Matos Galante & M. M. Moro, Type-safe Versioned Object Query Language; Journal of Universal Computer Science JUcs, 12(7), 938-957, 2006.
[31] E. Sciore, Using Annotations to Support Multiple Kinds of Versioning in an Object-Oriented Database System. ACM Trans. Database Syst. 16(3), 417-438, 1991.
[32] N.L. Sarda, P.V. Siva Prasada Reddy, Handling of Alternatives and Events in Temporal Databases, International Journal of Knowledge and Information Systems, Springer-Verlag 1( 3), 193-227, 1999.
[33] N.N. Dalvi, D. Suciu: Efficient query evaluation on probabilistic databases. VLDB J. 16(4): 523-544, 2007.
[34] C. Combi, Department of Computer Science of the University of Verona, 2011. (Personal communication)
[35] C. Combi, M. Gambini: Flaws in the Flow: The Weakness of Unstructured Business Process Modeling Languages Dealing with Data. OTM Conferences (1): 42-59, 2009.
[36] N. Trcka, W.M. P. van der Aalst, N. Sidorova: Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. CAiSE, 425-439, 2009.
[37] K. M. van Hee, Information systems engineering: a formal approach, Cambridge University Press New York, NY, USA, 1994.
[38] W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, J. Wainer: Proclets: A Framework for Lightweight Interacting Workflow Processes. Int. J. Cooperative Inf. Syst. (IJCIS) 10(4), 443-481, 2001.
[39] O. Habala, M. Paralic, V. Rozinajová, P. Bartalos, Semantically-Aided Data-Aware Service Workflow Composition. SOFSEM 2009: 317-328.
[40] C. Combi, M. Gozzi, J.M. Juárez, B. Oliboni, G. Pozzi: Conceptual Modeling of Temporal Clinical Workflows. TIME 2007, 70-81.

**Luca Anselma** received his PhD in Computer Science from Università di Torino in 2006. He is an assistant professor in Computer Science at the Università di Torino, Italy. His main research interests are in the areas of Temporal Reasoning, Temporal Databases, Model-based Diagnosis and Medical Informatics. He is the author of more than 30 papers in international journals, books and international refereed conferences.

**Alessio Bottrighi** took his Laurea degree in Computer Science at the Università del Piemonte Orientale, Italy and his PhD in Computer Science at the Università di Torino, Italy. He is an assistant professor at the Computer Science Department of Università del Piemonte Orientale. His main research interests include Medical informatics, Decision Support Systems, Temporal Databases, Case-Based Reasoning. On these topics, he has published more than 40 peer-reviewed papers in international journals, books and conference proceedings.

**Stefania Montani** received her PhD in Bioengineering from Università di Pavia in 2001. She is an assistant professor in Computer Science at the Università del Piemonte Orientale, in Alessandria, Italy. Her main research interests are in the areas of Temporal Databases, Case-Based Reasoning, Decision Support Systems and Temporal Reasoning. She is the author of more than 120 papers in international journals and international refereed conferences in Artificial Intelligence and Medical Informatics.

**Paolo Terenziani** received his Laurea degree in 1987 and his PhD in computer science in 1993 from Università di Torino, Italy. He is full professor in computer science with Dipartimento di Informatica, Università del Piemonte Orientale "Amedeo Avogadro", Alessandria, Italy. He is currently vice-Head of such Department. His research interests include artificial intelligence (knowledge representation and temporal reasoning), databases and computer science in medicine. He has published more than 100 papers on these topics in refereed journals and conference proceedings.